Scalable Parallel OPTICS Data Clustering Using Graph Algorithmic Techniques

Md. Mostofa Ali Patwary[1,†], Diana Palsetia[1], Ankit Agrawal[1],
Wei-keng Liao[1], Fredrik Manne[2], Alok Choudhary[1]

[1]Northwestern University, Evanston, IL 60208, USA [2]University of Bergen, Norway

[†]Corresponding author: mpatwary@eecs.northwestern.edu

ABSTRACT

OPTICS is a hierarchical density-based data clustering algorithm that discovers arbitrary-shaped clusters and eliminates noise using adjustable reachability distance thresholds. Parallelizing OPTICS is considered challenging as the algorithm exhibits a strongly sequential data access order. We present a scalable parallel OPTICS algorithm (POPTICS) designed using graph algorithmic concepts. To break the data access sequentiality, POPTICS exploits the similarities between the OPTICS algorithm and PRIM's Minimum Spanning Tree algorithm. Additionally, we use the disjoint-set data structure to achieve a high parallelism for distributed cluster extraction. Using high dimensional datasets containing up to a billion floating point numbers, we show scalable speedups of up to 27.5 for our OpenMP implementation on a 40-core shared-memory machine, and up to 3,008 for our MPI implementation on a 4,096-core distributed-memory machine. We also show that the quality of the results given by POPTICS is comparable to those given by the classical OPTICS algorithm.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering*; I.5.3 [**Pattern Recognition**]: Clustering—*Algorithms*; H.2.8 [**Database Management**]: Database Applications—*Data Mining*; G.1.0 [**Mathematics of Computing**]: Numerical Analysis—*Parallel Algorithms*

General Terms

Algorithms, Experimentation, Performance, Theory

Keywords

Density-based clustering, Minimum spanning tree, Union-Find algorithm, Disjoint-set data structure

1. INTRODUCTION

Clustering is a data mining technique that groups data into meaningful subclasses, known as *clusters*, such that it minimizes the intra-differences and maximizes inter-differences of these subclasses [21]. For the purpose of knowledge discovery, it identifies dense and sparse regions and therefore, discovers overall distribution patterns and correlations in the data. Based on the data properties or the task requirements, various clustering algorithms have been developed. Well-known algorithms include K-means [35], K-medoids [41], BIRCH [55], DBSCAN [20], OPTICS [5, 29], STING [52], and WaveCluster [48]. These algorithms have been used in various scientific areas such as satellite image segmentation [38], noise filtering and outlier detection [11], unsupervised document clustering [50], and clustering of bioinformatics data [36]. Existing clustering algorithms have been roughly categorized as partitional, hierarchical, grid-based, and density-based [26, 27]. OPTICS (Ordering Points To Identify the Clustering Structure) is a hierarchical density-based clustering algorithm [5]. The key idea of the density-based clustering algorithm such as OPTICS and DBSCAN is that for each data point in a cluster, the neighborhood within a given radius (ε), known as *generating distance*, has to contain at least a minimum number of points (*minpts*), i.e. the density of the neighborhood has to exceed some threshold [5, 20]. Additionally, OPTICS addresses DBSCAN's major limitation: the problem of detecting meaningful clusters in data of varying density.

OPTICS provides an overview of the cluster structure of a dataset with respect to density and contains information about every cluster level of the dataset. In order to do so, OPTICS generates a linear order of points where spatially closest points become neighbors. Additionally, for each point, a spacial distance (known as *reachability distance*) is computed which represents the density. Once the order and the reachability distances are computed using ε and $minpts$, we can query for the clusters that a particular value of ε' (known as *clustering distance*) would give where $\varepsilon' \leq \varepsilon$. The query is answered in linear time.

One example application of OPTICS, which requires high performance computing, is finding halos and subhalos (clusters) from massive cosmology data in astrophysics [34]. Other application domains include analyzing satellite images, X-ray crystallography, and anomaly detection [7]. However, OPTICS is challenging to parallelize as its data access pattern is inherently sequential. To the best of our knowledge, there has not been any effort yet to do so. Due to the similarities with DBSCAN, a natural choice for designing a parallel OPTICS could be one of the several master-slave based approaches [6, 13, 14, 17, 23, 54, 56]. However, in [44], we showed that these approaches incur high communication overhead between the master and slaves, and low parallel efficiency. As an alternative to these approaches we presented a parallel DBSCAN algorithm based on the disjoint-set data structure, suitable for massive data sets [44]. However, this approach is not directly applicable to OPTICS as DBSCAN produces a clustering result for a single set of density parameters, whereas OPTICS generates a linear order of the points that provides an overview of the cluster structure

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC 13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503255

for a wide range of input parameters. One key difference between these two algorithms from the viewpoint of parallelization is that in DBSCAN, after processing a point one can process its neighbors in parallel, whereas in OPTICS, the processing of the neighbors follows a strict order.

To overcome this challenge, we develop a scalable parallel OPTICS algorithm (POPTICS) using graph algorithmic concepts. POPTICS exploits the similarities between OPTICS and PRIM's Minimum Spanning Tree (MST) algorithm [46] to break the sequential access of data points in the classical OPTICS algorithm. The main idea is that two points should be assigned to the same cluster if they are sufficiently close (if at least one of them has sufficiently many neighbors). This relationship is transitive so a connected component of points should also be in the same cluster. If the distance bound is set sufficiently high, all vertices will be in the same cluster. As this bound is lowered, the cluster will eventually break apart forming sub-clusters. This is modeled by calculating a minimum distance spanning tree on the graph using an initial (high) distance bound (ε). Then to query the dataset for the clusters that an $\varepsilon' \leq \varepsilon$ would give, one has only to remove edges from the MST of weight more than ε' and the remaining connected components will give the clusters.

The idea of our POPTICS algorithm is as follows. Each processor computes a MST on its local dataset without incurring any communication. We then merge the local MSTs to obtain a global MST. Both steps are performed in parallel. Additionally, we extract the clusters directly from the global MST (without a linear order of the points) for any clustering distance, ε', by simply traversing the edges of the MST once in an arbitrary order, thus also enabling the cluster generation in parallel using the parallel disjoint-set data structure [42]. POPTICS shows higher concurrency for data access while maintaining a comparable time complexity and quality with the classical OPTICS algorithm. We note that MST-based techniques have been applied previously in cluster analysis, such as the single-linkage method that uses MST to join clusters by the shortest distance between them [25]. [22] and [31] also make the connection between OPTICS and PRIM's MST construction, but their proposed algorithms themselves do not exploit this idea to re-engineer OPTICS in order to implement it in a distributed environment or to achieve scalable performance.

POPTICS is parallelized using both OpenMP and MPI to run on shared-memory machines and distributed-memory machines, respectively. Our performance evaluation used a rich set of high dimensional data consisting of instances from real-world and synthetic datasets containing up to a billion floating point numbers. The speedups obtained on a shared-memory machine show scalable performance, achieving a speedup of up to 27.5 on 40 cores. Similar scalability results were observed for the MPI implementation on a distributed-memory machine with a speedup of 3,008 using 4,096 processors. In our experiments, we found that while achieving the scalability, POPTICS produces clustering results with comparable quality to the classical OPTICS algorithm.

The remainder of this paper is organized as follows. In Section 2, we describe the classical OPTICS algorithm. In Section 3, we propose the Minimum Spanning Tree based OPTICS algorithm along with a proof of correctness and complexity analysis. The parallel version, POPTICS is given in Section 4. We present our experimental methodology and results in Section 5 before concluding in Section 6.

2. THE OPTICS ALGORITHM

OPTICS is a hierarchical clustering algorithm that relies on a density notion of clusters [5, 20]. It is capable of detecting meaningful clusters in data of varying density by producing a linear order of points such that points which are spatially closest become neighbors in the order. OPTICS starts with adding an arbitrary point of a cluster to the order list and then iteratively expands the cluster by adding a point within the $\varepsilon-$neighborhood of a point in the cluster which is also closest to any of the already selected points. The process continues until the entire cluster has been added to the order. The process then moves on to the remaining clusters. Additionally, OPTICS computes the reachability distance for each point. This represents the required density in order to keep two points in the same cluster. Once the order and the reachability distances are computed, we can extract the clusters for any clustering distance, ε' where $\varepsilon' \leq \varepsilon$, in linear time. In the following we first define the notation used throughout the paper and then present a brief description of OPTICS based on [5].

Let X be the set of data points to be clustered. The neighborhood of a point $x \in X$ within a given radius ε (known as the *generating distance*) is called the *ε-neighborhood* of x, denoted by $N_\varepsilon(x)$. More formally, $N_\varepsilon(x) = \{y \in X | \text{DISTANCE}(x,y) \leq \varepsilon, y \neq x\}$, where $\text{DISTANCE}(x,y)$ is the distance function. A point $x \in X$ is referred to as a *core point* if its ε-neighborhood contains at least a minimum number of points ($minpts$), i.e., $|N_\varepsilon(x)| \geq minpts$. A point $y \in X$ is *directly density-reachable* from $x \in X$ if y is within the ε-neighborhood of x and x is a core point. A point $y \in X$ is *density-reachable* from $x \in X$ if there is a chain of points x_1,x_2,...,x_n, with $x_1 = x$, $x_n = y$ such that x_{i+1} is directly density-reachable from x_i for all $1 \leq i < n$, $x_i \in X$.

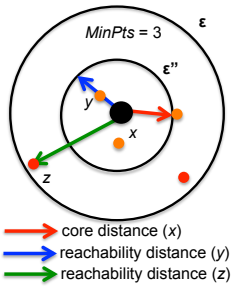


Figure 1: An example showing the *core distance* of x and the *reachability distances* of y and z with respect to x.

DEFINITION 2.1 (GENERATING DISTANCE). *The generating distance ε is the largest distance used to compute $N_\varepsilon(x)$ for each point $x \in X$.*

DEFINITION 2.2 (CORE DISTANCE). *The core distance,* CD, *of a point x is the smallest distance ε'' such that $|N_{\varepsilon''}(x)| \geq minpts$. If $|N_\varepsilon(x)| < minpts$, the core distance is* NULL.

DEFINITION 2.3 (REACHABILITY DISTANCE). *The reachability distance,* RD *of y with respect to another point x is either the smallest distance such that y is directly density reachable from x if x is a core point, or* NULL *if x is not a core point.*

$$\text{RD}(y) = \begin{cases} \text{NULL}, & \text{if } |N_\varepsilon(x)| < minpts \\ \text{MAX}\{\text{CD}(x), \text{DISTANCE}(x,y)\}, & \text{otherwise} \end{cases}$$

Figure 1 shows an example explaining the core distance of a point x and the reachability distance of y and z w.r.t. x.

The relationship between OPTICS and Minimum Spanning Tree computation is as follows. The OPTICS algorithm allows to extract clusters for different values of ε' (but always keeping $minpts$ fixed). It does so by constructing a minimum reachability distance spanning tree for the data points. Starting from an unprocessed point x such that $N_\varepsilon(x) \geq minpts$ it first picks and stores the point pair (or edge) (x,y) such that $y \in N_\varepsilon(x)$ where the reachability distance of y from x, $\text{RD}[y]$ is minimum. Then for any ε' where $\text{RD}[y] \leq \varepsilon'$, the points x and y will be in the same cluster as

Algorithm 1 The OPTICS algorithm. Input: A set of points X and the input parameters, generating distance, ε and the minimum number of points required to form a cluster, $minpts$. Output: An order of points, O, the core distances, and the reachability distances.

```
1: procedure OPTICS(X, ε, minpts, O)
2:     pos ← 0
3:     for each unprocessed point x ∈ X do
4:         mark x as processed
5:         N ← GETNEIGHBORS(x, ε)
6:         SETCOREDISTANCE(x, N, ε, minpts)
7:         O[pos] ← x; pos ← pos + 1
8:         RD[x] ← NULL
9:         if CD[x] ≠ NULL then
10:            UPDATE(x, N, Q)
11:            while Q ≠ empty do
12:                y ← EXTRACTMIN(Q)
13:                mark y as processed
14:                N′ ← GETNEIGHBORS(y, ε)
15:                SETCOREDISTANCE(y, N′, ε, minpts)
16:                O[pos] ← y; pos ← pos + 1
17:                if CD[y] ≠ NULL then
18:                    UPDATE(y, N′, Q)
```

long as $N_{\varepsilon'}(x) \geq minpts$. If $\varepsilon' < \text{RD}[y]$ then it is clear that x and y will be in different clusters as there cannot be a density-reachable path from x to y since y is the closest point to x. Thus it follows that for a given value ε' we can immediately determine if x and y should be in the same cluster. OPTICS then continues this process by repeatedly picking and storing the point z which is closest to the previously picked core points. In this way it builds a spanning tree very much like a minimum weight spanning tree in traditional graph theory. Once the tree is maximal one can query it for the clusters that a particular value of ε' would give where $\varepsilon' \leq \varepsilon$. The answer is obtained by removing any edge (x, y) where $\text{RD}(y) > \varepsilon'$ from the resulting spanning tree and returning the points in the remaining connected components as the clusters. In our work to parallelize the OPTICS algorithm, we exploit ideas from computing minimum weight spanning trees in parallel and also from how to compute connected components in parallel.

The pseudocode for the classical OPTICS algorithm is given in Algorithm 1. The algorithm computes the core distance and the reachability distance for each point and generates an order, O of all the points in X. It starts with an arbitrary point $x \in X$ and retrieves its ε-neighborhood, N (Line 5). It then computes the core distance, CD of x using the SETCOREDISTANCE function. If the ε-neighborhood of x does not contain at least $minpts$ points, then x is not a core point and therefore the SETCOREDISTANCE function sets $\text{CD}[x]$ to NULL. Otherwise, x is a core point and the SETCOREDISTANCE function finds the $minpts^{th}$ closest point to x in N and sets the distance to that point from x as $\text{CD}[x]$. This can easily be achieved by using a maximum priority queue of length equal to $minpts$ and traversing the points in N once.

The next step in OPTICS is to add x to the order, O (Line 7) and set the reachability distance of x, $\text{RD}[x]$ to NULL (Line 8) as it is the first point of the cluster added to the order or a noise point. We then check if the core distance of x is NULL, indicating that, x doesn't have sufficient neighbors to be a core point. In this case we continue to the next unprocessed point in X. Otherwise, we add (update if the reachability distance w.r.t. x is smaller) all the unprocessed neighbors in N into a (min) priority queue, Q for further processing using the UPDATE function (Line 10). The details of the UPDATE function are given in Algorithm 2. For each unprocessed point, $x' \in N$, the UPDATE function computes the reachability distance of x' w.r.t. x ($newD$). If the reachability distance of x' was NULL, we set $\text{RD}[x']$ to $newD$ and insert x' into Q for further processing. Otherwise, x' was already reached from other points and we check whether x is closer to x' compared to the earlier points. If so, we update $\text{RD}[x']$ to $newD$.

Algorithm 2 The UPDATE function. Input: A point x, its neighbors, N, and a priority queue, Q. Each element in Q stores two values, a point x' and its best reachability distance so far.

```
1: procedure UPDATE(x, N, Q)
2:     for each unprocessed point x′ ∈ N do
3:         newD = MAX(CD[x], DISTANCE(x′, x))
4:         if RD[x′] = NULL then
5:             RD[x′] ← newD
6:             INSERT(Q, x′, newD)
7:         else if newD < RD[x′] then
8:             RD[x′] ← newD
9:             DECREASE(Q, x′, newD)
```

The next step in Algorithm 1 is to process each point $y \in Q$ (Line 11-18) in the same way as discussed above (Line 4-10) for x. As Q is a minimum priority queue, each point y extracted from Q is the closest neighbor of the already processed points belonging to the current cluster. Note that while processing the points in Q, new neighbors might be added or the reachability distance of the unprocessed points in Q might be updated, which essentially changes the order of the points in Q. When Q is empty, the entire cluster has been explored and added to the order, O. The algorithm then continues to the next unprocessed point in X.

The computational complexity of Algorithm 1 is $O(n * runtime\ of\ an\ \varepsilon\text{-}neighborhood\ query)$, where n is the number of points in X. The retrieval of ε-neighborhood of a point (GETNEIGHBORS function) in Algorithm 1 is known as a *region-query* with center x and generating distance ε. This requires a linear scan of the entire datasets, therefore, the complexity of Algorithm 1 is $O(n^2)$. But, if spatial indexing (for example, using a *kd-tree* [28] or an *R*-tree* [8]) is used for serving the region-queries (GETNEIGHBORS function), the complexity reduces to $O(n \log n)$ [9].

Algorithm 3 The ORDERTOCLUSTERS function. Input: An order of points, O and clustering distance, ε'. Output: Clusters in CID.

```
1: procedure ORDERTOCLUSTERS(O, ε′)
2:     id ← 0
3:     for each point x ∈ O do
4:         if RD[x] > ε′ then
5:             if CD[x] ≤ ε′ then
6:                 id ← id + 1
7:                 CID[x] = id
8:             else
9:                 CID[x] = NOISE
10:        else
11:            CID[x] = id
```

Once the order, core distances, and the reachability distances are computed by Algorithm 1, any density-based clusters of clustering distance, ε', ranging from 0 to ε, can be extracted in linear time. Algorithm 3, ORDERTOCLUSTERS provides the pseudocode for extracting the clusters for an ε'. The idea is that two points x and y belong to the same cluster if one, say x, is directly density reachable (w.r.t. ε') from the other one, y (a core point), that is, $\text{RD}[x] \leq \varepsilon'$ which ensures that $\text{CD}[y] \leq \varepsilon'$. Since the closest points in X are grouped together in the order O, ORDERTOCLUSTERS finds the groups (each group is a cluster) satisfying this criteria. However, for the first point x of a cluster in O, $\text{RD}[x]$ is greater than ε', but x is a core point, that is, $\text{CD}[x] \leq \varepsilon'$ (Line 4-5). We therefore begin a

new cluster (Line 6-7) and keep adding the following points, say y, in O (Line 11) as long as y is directly density reachable from any of the previously added core points, say z, in the same cluster, that is, $\text{RD}[y] \leq \varepsilon'$, which implies that $\text{CD}[z] \leq \varepsilon'$. Any point not part of a cluster (not reachable w.r.t. ε') is declared as a NOISE point (Line 9). The process continues until all points in O are scanned. Since we traverse the order O once to extract all the clusters and noise points, the complexity of Algorithm 3 is linear.

Note that deriving clusters in such a way in OPTICS and running DBSCAN with the chosen clustering distance ε' yield the same clustering result on the core points of a dataset. The assignment of non-core points to neighboring clusters is non-deterministic both in DBSCAN and in OPTICS. However, to obtain a hierarchical clustering using DBSCAN requires multiple runs of the expensive clustering algorithm. This method also incurs a huge memory overhead to store the cluster memberships for many different values of the input parameter, ε'. On the contrary, OPTICS executes the expensive clustering algorithm once for a larger value of ε', the generating distance ε to store the structure of the datasets and later, extracts the clusters in linear time for any value of ε', where $\varepsilon' \leq \varepsilon$.

3. DESIGN FOR SCALABILITY

The major limiting factor when parallelizing the OPTICS algorithm is that it exhibits a strongly inherent sequential data access order always processing the closest point when producing the order and the reachability distances. To break this sequentiality, we present a new OPTICS algorithm which exploits the similarities between OPTICS and PRIM's Minimum Spanning Tree (MST) algorithm.

As mentioned before, [22] and [31] also make the connection between OPTICS and PRIM's MST construction, but their proposed algorithms themselves do not exploit this idea to re-engineer OPTICS in order to implement it in a distributed environment or to achieve scalable performance. Moreover, the algorithms assume that the reachability distances between two points are symmetric, which in reality is not the case for classical OPTICS. Additionally, [22] is computationally expensive as it does not use any input parameter (e.g. ε). In contrast to our approach which extract clusters directly from the MST in parallel, these algorithms compute the order from the MST and the clusters from the order, which make the whole process more sequential.

In the following, we provide a brief overview of the PRIM's MST algorithm [46], followed by the details of our new MST-based OPTICS algorithm.

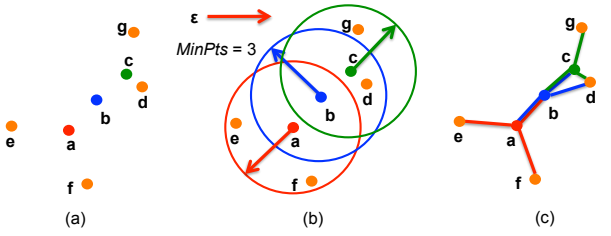


Figure 2: An example showing the similarities between OPTICS and PRIM's MST algorithm. (a) The data points, (b) The ε neighborhood of point a, b and c in OPTICS, and (c) The adjacent edges of vertices a, b and c in PRIM's MST algorithm. As can be seen, starting from a, the processing order in both cases are the same, $a \rightarrow b \rightarrow c \rightarrow \ldots$ as b and c are the closest point (vertex) of a and b, respectively.

3.1 Prim's Minimum Spanning Tree

A subgraph T of a graph $G = (V, E)$, where V and E denote the set of vertices and edges of graph G, respectively, is a *spanning tree* of G if it is a tree and contains each vertex $v \in V$. A Minimum Spanning Tree (MST) is a spanning tree of minimum weight, where weight in our setting is the reachability distance between two points. PRIM's algorithm [46] is a greedy approach to find an MST of a given graph. The algorithm starts with adding an arbitrary vertex into the MST. It then iteratively grows the current MST by inserting a vertex closest to the vertices already in the current MST. The process continues until all the vertices are added into the tree. If the graph contains multiple components, then PRIM's algorithm finds one MST for each component. The set of MSTs are known as *Minimum Spanning Forest* (MSF). Throughout the paper we use MST and MSF interchangeably. The complexity of PRIM's algorithm is $O(|E|log|V|)$ if implemented with a simple binary heap data structure and an adjacency list representation for G.

As discussed before, PRIM's approach to continuously increase the MST, one edge at a time, is very similar to the OPTICS algorithm. In the context of OPTICS, the vertices are analogous to the points in our spatial dataset and edge weights are analogous to the reachability distances. We therefore use them interchangeably throughout the paper. While expanding the MST, PRIM's algorithm considers the adjacent vertices whereas in OPTICS, all points in the ε-neighborhood are considered as adjacent points. Figure 2 is an example showing the similarities between OPTICS and PRIM's MST algorithm. The ε-neighborhoods of point a, b, and c are shown in Figure 2(b) and the corresponding edges for them are shown in Figure 2(c). The color of the circles and the color of the edges match with the color of the points explored.

3.2 A New MST-based OPTICS Algorithm

As discussed above, the OPTICS approach to hierarchical density based clustering consists of two stages. First, Algorithm 1 to compute the order, core distances, and reachability distances using the generating distance, ε and minimum number of points, $minpts$. This is followed by Algorithm 3 for extracting clusters from these values using the clustering distance, ε', where $0 \leq \varepsilon' \leq \varepsilon$. Similarly our Minimum Spanning Tree (MST) based OPTICS has two corresponding stages, (i) MOPTICS (Algorithm 4) to compute the MST and core distances, and (ii) extracting clusters (Algorithm 5) from the already computed MST and core distances. We also show that using the MST, one can compute the reachability distances and the order of points in linear time.

Algorithm 4 Compute the Minimum Spanning Trees based on Prim's algorithm. Input: A set of points X and the input parameters, ε and $minpts$. Output: The Minimum Spanning Trees, T.

```
1:  procedure MOPTICS(X, ε, minpts, T)
2:      for each unprocessed point x ∈ X do
3:          mark x as processed
4:          N ← GETNEIGHBORS(x, ε)
5:          SETCOREDISTANCE(x, N, ε, minpts)
6:          if CD[x] ≠ NULL then
7:              MODUPDATE(x, N, P)
8:              while P ≠ empty do
9:                  (u, v, w) ← EXTRACTMIN(P)
10:                 T ← T ∪ (u, v, w)
11:                 mark u as processed
12:                 N' ← GETNEIGHBORS(u, ε)
13:                 SETCOREDISTANCE(u, N', ε, minpts)
14:                 if CD[u] ≠ NULL then
15:                     MODUPDATE(u, N', P)
```

The pseudocode of the first stage to compute the MST and core distances is given in Algorithm 4, denoted by MOPTICS. The algorithm is similar to the classical OPTICS (Algorithm 1), but instead of computing the reachability distances and order, it computes the MST (similar to PRIM's Minimum Spanning Tree Algorithm). However, both algorithms compute the core distances. MOPTICS starts with an arbitrary point $x \in X$ and retrieves its ε-neighborhood, N using the GETNEIGHBORS function (Line 4). We then compute the core distance of x using the SETCOREDISTANCE function (Line 5). Note that we use the same GETNEIGHBORS and SETCOREDISTANCE functions as were used in the classical OPTICS algorithm (Algorithm 1). If x is not a core point, we continue to the next unprocessed point, otherwise, we add or update each unprocessed neighbor in N into a minimum priority queue, P, using a modified UPDATE function, named MODUPDATE.

For each neighbor $x' \in N$, MODUPDATE computes the reachability distance of x' w.r.t. x and adds or decreases its value in P depending on if it already existed in P or if the new reachability distance is smaller than the earlier computed one. Note that although ideally UPDATE and the MODUPDATE functions are identical, MODUPDATE additionally stores the source point (x in this case) from which the point x' has been reached. This is to achieve a memory efficient parallel MOPTICS (discussed in the next section) as otherwise each process core requires a vector of reachability distances of length equal to the total number of points in X.

The next step (Line 8-15) in Algorithm 4 is to process all the points in the priority queue, P, in the same way as discussed above for x (Line 3-7). Additionally, we add the extracted triples (u, v, w) in Line 9 as an edge, $v \rightarrow u$ with weight w (as point u was reached from v with reachability distance w) into the MST, T, in Line 10. Similar to classical OPTICS, while processing the points in P, more points might be added into P (until the entire cluster is explored). Since a point can be inserted into P at most once throughout the computation, the edges in T do not form any cycle.

For each point, $x \in X$, MOPTICS uses all the points in the ε-neighborhood, N, of x as the adjacent points (similar to PRIM's algorithm which considers the adjacent vertices while processing a vertex) and the reachability distance of each point $x' \in N$ from x is considered as the weight of the edge $x \rightarrow x'$ in the MST. The complexity of MOPTICS is $O(n \log n)$, identical to OPTICS.

Note that given the MST, one can easily compute the order and the reachability distances from the MST produced by MOPTICS in linear time (w.r.t. to the number of edges in the MST). The idea is to rerun the MOPTICS algorithm, but instead of computing the core distances (as already computed), we compute the reachability distances and order using only the edges in the MST. This process also does not need the expensive ε-neighborhood query (GETNEIGHBORS function) as the neighbors can be found from the MST itself. This computation only takes a fraction of the time compared to OPTICS and MOPTICS. Due to space consideration, we only present the results in the experiments section.

We now present the second stage of our MST-based OPTICS algorithm, named MSTTOCLUSTERS, to extract the clusters directly from the MST for any clustering distance ε', where $0 \leq \varepsilon' \leq \varepsilon$. This is achieved by removing any edge $x \rightarrow y$ with weight $w > \varepsilon'$ (hence $\text{RD}(y) > \varepsilon'$) from the MST and returning the points in the remaining connected components as the clusters. The MSTTOCLUSTERS function uses the *disjoint-set data structure* [24, 43] for this purpose. Below we briefly introduce the data structure and how it works.

The disjoint-set data structure defines a mechanism to maintain a dynamic collection of non-overlapping sets of points. The data structure comes with two main operations: FIND and UNION. The FIND operation determines to which set a given point belongs, while the UNION operation joins two existing sets. Each set is identified by a representative, x, which is usually some point of the set. The underlying data structure of each set is typically a rooted tree represented by a parent pointer, $\text{PAR}(x)$ for each point $x \in X$; the root satisfies $\text{PAR}(x) = x$ and is the representative of the set. The output of the $\text{FIND}(x)$ operation is the root of the tree containing x. $\text{UNION}(x, y)$ merges the two trees containing x and y by changing the parent pointer of one root to the other one. To do this, the $\text{UNION}(x, y)$ operation first calls two find operations, $\text{FIND}(x)$ and $\text{FIND}(y)$. If they return the same root (i.e. x and y are in the same set), no merging is required. But if the returned roots are different, say r_x and r_y, the UNION operation sets $\text{PAR}(r_x) = r_y$ or $\text{PAR}(r_y) = r_x$. Note that this definition of the UNION operation is slightly different from its standard definition which requires that x and y belong to two different sets before calling UNION. There exist many different techniques to improve the performance of the UNION operation. In this paper, we have used the empirically best known UNION technique (a lower indexed root points to a higher indexed root), known as REM's algorithm with the splicing compression technique. Details on these can be found in [43].

The pseudocode of MSTTOCLUSTERS is given in Algorithm 5. The idea is that two vertices u and v connected by an edge with weight w belong to the same cluster if $w \leq \varepsilon'$. For each point $x \in X$, MSTTOCLUSTERS starts by creating a new set by setting the parent pointer to itself (Line 2-3). We then go through each edge (u, v, w) in the MST, T (Line 4-6). We check whether the edge weight $w \leq \varepsilon'$. If so, then u is density reachable from v with reachability distance w, and v is core point with core distance, $\text{CD}[v] \leq \varepsilon'$ as $\text{CD}[v] \leq \text{RD}[u]$. Therefore, u and v should belong to the same cluster. We therefore perform a UNION operation (Line 6) of the trees containing u and v. At the end of the MSTTOCLUSTERS algorithm, a singleton tree containing only one point is a NOISE point whereas all points in a tree of size more than one belong to the same cluster.

Algorithm 5 The MSTTOCLUSTERS function. Input: The Minimum Spanning Trees, T and the clustering distance, ε'. Output: Clusters in CID.

```
1: procedure MSTTOCLUSTERS(T, ε')
2:     for each point x ∈ X do
3:         PAR(x) ← x
4:     for each edge (u, v, w) ∈ T do
5:         if w ≤ ε' then
6:             UNION(u, v)
```

THEOREM 3.1. *Algorithm 3 (*ORDERTOCLUSTERS*) and Algorithm 5 (*MSTTOCLUSTERS*) produce identical clusters.*

Due to space consideration, we only outline the proof. Given the order of points, O generated by OPTICS, ORDERTOCLUSTERS adds a point, u to a cluster if the reachability distance, $\text{RD}[u] \leq \varepsilon'$. This implies that u is reachable from a core point v with core distance $\text{CD}[v] \leq \varepsilon'$ as $\text{CD}[v] \leq \text{RD}[u]$. Any point following u in O with reachability distance less than or equal to ε' belongs to the same cluster, S. Therefore, ORDERTOCLUSTERS keeps adding them to S. As MOPTICS doesn't have the order, but it rather stores the reachability distance of u as an edge weight w along with the point v from which it has been reached in the MST. Therefore, for each edge, if the edge weight $w \leq \varepsilon'$ (thus $\text{RD}[u] \leq \varepsilon'$), then u and v must belong to the same cluster, as is done in MSTTOCLUSTERS.

Note that as MSTTOCLUSTERS can process the edges in T in an arbitrary order, it follows that, it is highly parallel in nature. However, this stage takes only a fraction of the time taken compared to OPTICS and MOPTICS as will be shown in the experiments

Algorithm 6 The parallel OPTICS algorithm on a shared memory computer (POPTICSS) using p threads. Input: A set of points X and the input parameters, ε and $minpts$. Let X be divided into p equal disjoint partitions $X_1, X_2, X_3, \ldots, X_p$, each assigned to one of the p threads. Output: The Minimum Spanning Trees, T.

```
1:  procedure POPTICSS(X, ε, minpts, T)
2:      for t = 1 to p in parallel do   ▷ Stage: Local computation
3:          for each unprocessed point x ∈ X_t do
4:              mark x as processed
5:              N ← GETNEIGHBORS(x, ε)
6:              SETCOREDISTANCE(x, N, ε, minpts)
7:              if CD[x] ≠ NULL then
8:                  MODUPDATE(x, N, P_t)
9:                  while P_t ≠ empty do
10:                     (u, v, w) ← EXTRACTMIN(P_t)
11:                     Q ← INSERT(u, v, w) in critical
12:                     if u ∈ X_t then
13:                         mark u as processed
14:                         N' ← GETNEIGHBORS(u, ε)
15:                         SETCOREDISTANCE(u, N', ε, minpts)
16:                         if CD[u] ≠ NULL then
17:                             MODUPDATE(u, N', P_t)
18:     for each point x ∈ X in parallel do   ▷ Stage: Merging
19:         PAR(x) ← x
20:     while Q ≠ empty do
21:         (u, v, w) ← EXTRACTMIN(Q)
22:         if UNION(u, v) = TRUE then
23:             T ← T ∪ (u, v, w)
```

section. We therefore omit the discussion on the parallelization of MSTTOCLUSTERS and only present how to parallelize the first stage, MOPTICS. However, it is worth noting that parallelization of MSTTOCLUSTERS can be achieved easily using our prior work on PARALLELUNION algorithm, for both shared and distributed memory computers [37, 42]. The idea behind the PARALLELUNION operation on a shared memory computer is that the algorithm uses a separate lock for each point. A thread wishing to set the parent pointer of a root r_1 to r_2 during a UNION operation would then have to acquire r_1's lock before doing so. More details on parallel UNION using locks can be found in [42]. However, in distributed memory computers, as the memory is not shared among the processors, message passing is used instead of locks and only the owner of a point is allowed to change the parent pointers. Details are available in [37].

4. THE PARALLEL OPTICS ALGORITHM

We parallelize the OPTICS algorithm by exploiting ideas for how to compute Minimum Spanning Trees in parallel. The key idea of our parallel MST-based OPTICS (denoted by POPTICS) is that each process core first runs the sequential MOPTICS algorithm (Algorithm 4) on its local data points to compute local MSTs in parallel. We then perform a parallel merge of the local MSTs to obtain the global MST. Note that similar ideas have been used in the graph setting for shared address space and GPUs [40, 47].

4.1 POPTICS on Shared Memory

The details of parallel OPTICS on shared memory parallel computers (denoted by POPTICSS) are given in Algorithm 6. The data points X are divided into p partitions $\{X_1, X_2, \ldots, X_p\}$ (one for each of the p threads running in parallel) and each thread t owns partition X_t. We divide the algorithm, POPTICSS, into two segments, *local computation* (Line 2-17) and *merging* (Line 18-23). Local computation is similar to sequential MOPTICS but each thread t processes only its own data points X_t instead of X and also operates on its own priority queue, P_t while processing the neighbors. Additionally, when we extract a point u (reached from v with reachability distance, w, Line 10), we perform the following two things: (i) We add the edge (u, v, w) into a minimum priority queue, Q (Line 11), shared among the threads (therefore is a *critical* statement) for further processing in the merging stage, and (ii) we check if $u \in X_t$ (Line 12) to make sure thread t processes only its own points as the GETNEIGHBORS function (Line 5 and 14) returns both local and non-local points as all points are in the commonly accessible shared memory.

It is very likely that the union of the local MSTs found by each thread in the local computation contains redundant edges (in particular those connecting local and non-local points), thus giving rise to cycles. We therefore need one additional *merging* stage (Line 18-23) to compute the *global* MST from the local MSTs, stored in Q. One can achieve this using either PRIM's MST algorithm [46] or KRUSKAL's MST algorithm [30]. However, PRIM's algorithm requires an additional computation to obtain the adjacency list computed from the edges in Q. We therefore use KRUSKAL's algorithm in the merging stage. KRUSKAL's algorithm also needs the edges in Q to be in a sorted order, but this is achieved as a by-product as Q is a priority queue and the edges were inserted in a *critical* statement. KRUSKAL's algorithm uses the disjoint-set data structure [18, 51] as discussed above. It first creates a new set for each point, $x \in X$ (Line 18-19) in parallel. It then iteratively extracts the top edge, (u, v, w) from Q and tries to perform a UNION operation of the sets containing u and v. If they are already in the same set, we continue to the next edge. Otherwise, the UNION operation returns TRUE and we add the edge (u, v, w) into the global MST, T. Although the merging stage on shared memory implementation is mostly sequential, it takes only a very small portion of the total time taken by POPTICSS. This is because merging does not require any communication and operates only on the edges in the local MSTs.

Thus, given the global MST, one can compute the clusters for any clustering distance ε' using MSTTOCLUSTERS function (Algorithm 5). Note that the MST computed by parallel OPTICS, POPTICSS could be different than the MST computed by sequential MST-based OPTICS, MOPTICS. This is mainly because of the following three reasons: (i) The processing sequence of the points along with the starting points is different, (ii) The reachability distance between two neighbor points x and y are not symmetric (although $\text{RD}[x]$ w.r.t. y can be computed using $\text{RD}[y]$ w.r.t. x and $\text{CD}[y]$), and (iii) A *border* point (neither a core point nor a noise point, but falls within the ε-neighborhood of a core point) that falls within the boundary of two clusters will be taken by the first one which reaches the point. Therefore, to evaluate the quality of the clusters obtained from POPTICSS compared to the clusters given by classical OPTICS, we employ a well known metric, called *Omega-Index* [16], designed for comparing clustering solutions [39, 53]. The Omega-Index is based on how the pairs of points have been clustered. Two solutions are in agreement on a pair of points if they put both points into the same cluster or each into a different cluster. Thus the Omega-Index is computed using the observed agreement adjusted by the expected agreement divided by the maximum possible agreements adjusted by the expected agreement. The score ranges from 0 to 1, where a value of 1 indicates that the two solutions match. More details can be found in [16, 39, 53].

4.2 POPTICS on Distributed Memory

The details of parallel OPTICS (POPTICS) on distributed memory parallel computers (denoted by POPTICSD) are given in Algorithm 7. Similar to POPTICSS and traditional parallel algorithms, we assume that the data points X have been equally partitioned into

p partitions $\{X_1, X_2, \ldots, X_p\}$ (one for each processor) and each processor t owns X_t only. A point x is a *local point* on processor t if $x \in X_t$, otherwise x is a *remote point*. Since the memory is distributed, any partition $X_i \neq X_t$, $1 \leq i \leq p$ is invisible to processor t (in contrast to PopticsS which uses shared memory). We therefore need the GetLocalNeighbors (Line 4) and GetRemoteNeighbors (Line 5) functions to get the local and remote points, respectively. Note that retrieving the remote points requires communication with other processors. Instead of calling GetRemoteNeighbors for each local point during the computation, we take advantage of the ε parameter and gather all possible remote neighbors in one step before the start of the algorithm. In the Optics algorithm, for any given point x, we are only interested in the neighbors that fall within the generating distance ε of x. Therefore, we extend the bounding box of X_t by a distance, ε, in every direction in each dimension and query other processors with the extended bounding box to return their local points that fall in it. Thus, each processor t has a copy of the remote points X'_t that it requires for its computation. We consider this step as a preprocessing step (named *gather-neighbors*). Our experiments show that gather-neighbors takes only a limited time compared to the total time. Thus, the GetRemoteNeighbors function returns the remote points from the local copy, X'_t without communication.

Algorithm 7 The parallel Optics algorithm on a distributed memory computer (PopticsD) using p processors. Input: A set of points X and the input parameters, ε and $minpts$. Let X be divided into p equal disjoint partitions $\{X_1, X_2, \ldots, X_p\}$ for the p running processors. Each processor t also has a set of remote points, X'_t stored locally to avoid communication during local computation. Output: The Minimum Spanning Trees, T.

```
1:  procedure PopticsD(X, ε, minpts)
2:      for each unprocessed point x ∈ X_t do        ▷ Stage: Local computation
3:          mark x as processed
4:          N_l ← GetLocalNeighbors(x, ε)
5:          N_r ← GetRemoteNeighbors(x, ε)
6:          N ← N_l ∪ N_r
7:          SetCoreDistance(x, N, ε, minpts)
8:          if Cd[x] ≠ Null then
9:              ModUpdate(x, N, P_t)
10:             while P_t ≠ empty do
11:                 (u, v, w) ← ExtractMin(P_t)
12:                 Q_t ← Insert(u, v, w)
13:                 if u ∈ X_t then
14:                     mark u as processed
15:                     N'_l ← GetLocalNeighbors(u, ε)
16:                     N'_r ← GetRemoteNeighbors(u, ε)
17:                     N' ← N'_l ∪ N'_r
18:                     SetCoreDistance(u, N', ε, minpts)
19:                     if Cd[u] ≠ Null then
20:                         ModUpdate(u, N', P_t)
21:     round ← log_2(p) − 1                        ▷ Stage: Merging
22:     for i = 0 to round do
23:         if t mod 2^i = 0 then                   ▷ check if t participates
24:             if t mod 2^(i+1) = 0 then           ▷ t is receiver
25:                 t' ← t + 2^i                    ▷ t' is the sender
26:                 receive Q_t' from P_t'
27:                 Q_t ← Kruskal(Q_t ∪ Q_t')
28:             else
29:                 t' ← t − 2^i                    ▷ t' is receiver
30:                 send Q_t from P_t'              ▷ t is sender
```

Similar to PopticsS, PopticsD also has two stages, *local computation* (Line 2-20) and *merging* (Line 21-30). During the local computation, we compute the local neighbors, N_l (Line 4) and remote neighbors, N_r (Line 5) for each point x. Based on these we then compute the core distance using the SetCoreDistance function. The rest of the local computation is similar to PopticsS except that the tree edges extracted from the priority queue, P_t (Line 11) are inserted into a local priority queue, Q_t, instead of a shared queue, Q, as was done in PopticsS.

As the local MSTs found by the local computation are distributed among the process cores, we need to gather and merge these local MSTs to remove any redundant edges (thus breaking cycles) to obtain the *global* MST. To do this, we perform a *pairwise-merging* in the merging stage (Line 21-30). To simplify the explanation, we assume that p is a multiple of 2. The merging stage then runs in $\log_2(p)$ rounds (Line 22) following the structure of a binary tree with p leaves. In each merging operation, the edges of two local MSTs are gathered on one processor. This processor then computes a new local MST using Kruskal's algorithm on the gathered edges. After the last round of merging, the global MST will be stored on processor 0.

We use Kruskal's algorithm in the merging stage even though Boruvka's MST algorithm [15] is inherently more parallel than Kruskal's. This is because Boruvka's algorithm requires edge contractions, which in distributed memory would require more communication especially when the contracted edge spans different processors. Since we get an ordering of the edges as a by-product of the main algorithm, this makes Kruskal's algorithm more competitive for the merging stage.

However, for the merging stage, we implemented a couple of variations to improve the performance and memory scalability, but we only give an outline due to space considerations. In each of the $\log_2 p$ rounds in the merging, we traverse the entire graph (local MSTs) once, thus the overhead is proportional to the number of rounds. We therefore tried to terminate the pairwise-merging after the first few rounds and then gather the rest of the merged local MSTs on process core 0 to compute the global MST. Another technique we implemented was to exclude the edges from the local MSTs during the pairwise-merging using Boruvka's concept [15]. For each point x, the lightest edge connecting x will definitely be part of the global MST. We also considered a hybrid version of these approaches.

5. EXPERIMENTAL RESULTS

We first present the experimental setup used for both the sequential and the shared memory Optics algorithms. The setup for the distributed memory algorithm is presented later.

For the sequential and shared memory experiments we used a Dell computer running GNU/Linux and equipped with four 2.00 GHz Intel Xeon E7-4850 processors with a total of 128 GB memory. Each processor has ten cores. Each of the 40 cores has 48 KB of L1 and 256 KB of L2 cache. Each processor (10 cores) shares a 24 MB L3 cache. All algorithms were implemented in C++ using OpenMP and compiled with gcc (version 4.7.2) using the -O2 flag.

Our testbed consists of 18 datasets, which are divided into three categories, each with six datasets. The first category, called *real-world*, representing alphabets and textures for information retrieval, has been collected from Chameleon (*t4.8k*, *t5.8k*, *t7.10k*, and *t8.8k*) [2] and CUCIS (*edge* and *texture*) [1]. The other two categories, *synthetic-random* and *synthetic-cluster*, have been generated synthetically using the IBM synthetic data generator [4, 45]. In the synthetic-random datasets (*r50k*, *r100k*, *r500k*, *r1m*, *r1.5m*, and *r1.9m*), points in each dataset have been generated uniformly at random. In the synthetic-cluster datasets (*c50k*, *c100k*, *c500k*, *c1m*, *c1.5m*, and *c1.9m*), first a specific number of random points are taken as different clusters, points are then added randomly to these clusters. The testbed contains up to 1.9 million data points and each data point is a vector of up to 20 dimensions. Table 1 shows struc-

tural properties of the dataset. In the experiments, the two input parameters (ε and $minpts$) shown in the table have been chosen carefully to obtain the order, core distances, and reachability distances in a reasonable time. Higher value of ε increases the time taken for the experiments while the number of clusters and noise points are reduced. Higher value of $minpts$ increases the noise counts. We also select a clustering distance, ε' to extract a fair number of clusters and noise points from the order or the MST.

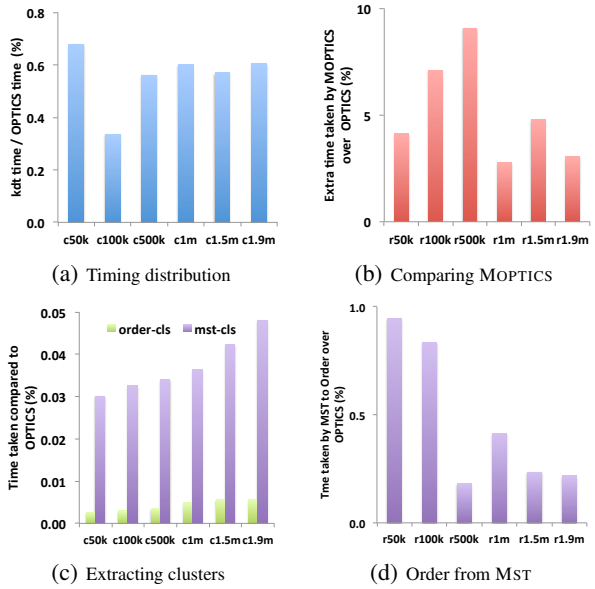
Table 1: Structural properties of the testbed (real-world, synthetic-random, and synthetic-cluster) and the time taken by the OPTICS and the MOPTICS algorithms. d denotes the dimension of each point. The last three columns show the resulting number of clusters and noise points using an ε'.

				min	Time (sec.)		Sample extraction		
Name	Points	d	ε	pts	OPTICS	MOPTICS	ε'	Clusters	Noise
$t4.8$	24,000	2	30	20	1.27	1.51	10	18	2,026
$t5.8$	24,000	2	30	20	1.98	2.40	10	27	1,852
$t7.10$	30,000	2	30	20	1.18	1.18	10	84	4,226
$t8.8$	24,000	2	30	20	1.14	1.23	10	165	9,243
$edge$	336,205	18	2	3	569.53	574.03	1.5	1,368	83,516
$texture$	371,595	20	2	3	1,124.93	1,183.44	1.5	2,058	310,925
$r50k$	50,000	10	120	5	8.47	8.82	100	1,058	42,481
$r100k$	100,000	10	120	5	22.31	23.89	100	883	33,781
$r500k$	500,000	10	120	5	694.71	757.77	100	2	1,161
$r1m$	1,000,000	10	80	5	813.15	835.65	60	10,351	948,867
$r1.5m$	1,500,000	10	80	5	2479.70	2,598.80	60	7,809	326,867
$r1.9m$	1,900,000	10	80	5	3680.21	3,792.77	60	8,387	368,917
$c50k$	50,000	10	120	5	11.52	14.05	25	51	3,095
$c100k$	100,000	10	120	5	22.49	27.57	25	103	6,109
$c500k$	500,000	10	120	5	119.99	142.80	25	512	36,236
$c1m$	1,000,000	10	80	5	226.11	275.97	25	1,022	64,740
$c1.5m$	1,500,000	10	80	5	331.07	405.12	25	1,543	102,818
$c1.9m$	1,900,000	10	80	5	431.67	526.30	25	1,949	135,948

5.1 OPTICS vs. MOPTICS

As discussed in Section 2, to reduce the running time of the OPTICS algorithm from $O(n^2)$ to $O(n \log n)$, spatial indexing (*kd-tree* [28] or *R*-tree* [8]) is commonly used [9]. In all of our implementations, we used *kd-trees* [28] and therefore obtain the reduced time complexities. Moreover, the *kd-tree* gives a geometric partitioning of the data points, which we use to divide the data points equally among the cores in the parallel OPTICS algorithm. However, there is an overhead in constructing the *kd-tree* before running the OPTICS algorithms. Figure 3(a) shows a comparison of the time taken by the construction of the *kd-tree* over the OPTICS algorithm in percent for the synthetic-cluster datasets. As can be seen, constructing the *kd-tree* takes only a fraction of the time (0.33% to 0.68%) taken by the OPTICS algorithm. We found similar results for the synthetic-random datasets (0.07% to 0.93%). However, these ranges are somewhat higher (0.06% to 3.23%) for the real-world dataset. This is because each real-world dataset consists of a small number of points, and therefore, the OPTICS algorithm takes less time compared to the other two categories. It should be noted that we have not parallelized the construction of the *kd-tree* in this paper, we therefore do not consider the timing of the construction of the *kd-tree* in the following discussion.

Figure 3(b) presents the performance of MOPTICS (Algorithm 4) compared to OPTICS (Algorithm 1) on synthetic random datasets. The raw run-times taken by MOPTICS and OPTICS are provided in Table 1. As can be seen, MOPTICS takes on average 5.16% (range 2.77%-9.08%) more time compared to the time taken by OPTICS on synthetic-random datasets. This is because, MOPTICS stores the reachability distance in a map (instead of a vector as in OPTICS) and therefore, retrieving the distances to update takes additional time. In OPTICS, this is achieved in constant time as it uses a vector (Line 7 in Algorithm 2). This additional computation is needed



(a) Timing distribution (b) Comparing MOPTICS

(c) Extracting clusters (d) Order from MST

Figure 3: Performance of (a) the construction of *kd-tree*, (b) MOPTICS (Algorithm 4), (c) extracting the clusters, and (d) computing the order from MST, compared to OPTICS.

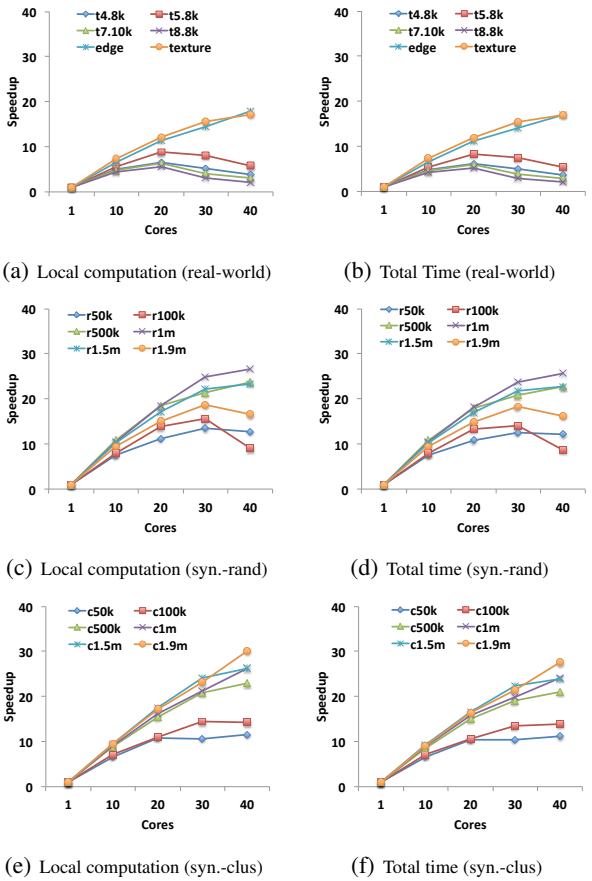
in the parallel OPTICS algorithms, otherwise, each core requires a vector of length equal to the number of total points among the cores. We observe similar performance for the real-world datasets (average 9.09%, range 0.12%-21.34%). This value is higher for the synthetic-cluster datasets with an average of 21.64% (range 19.01%-22.59%) as we observed that the number of neighbor updates is much higher compared to the other categories.

Figure 3(c) shows the time taken to extract the clusters from the order and the MST (denoted by *order-cls* and *mst-cls*, respectively) compared to the classical OPTICS algorithm for synthetic-cluster datasets. The clustering distances ε', used in the experiments can be found in Table 1. As can be seen, both *order-cls* and *mst-cls* take a small fraction of time (maximum 0.01% and 0.05% respectively) compared to OPTICS, and are comparable to each other. The relative maximum time spent in *order-cls* is unchanged for the synthetic-random and real-world datasets, while the maximum time spent in *mst-cls* is 0.04% and 0.19%, respectively. Note that OPTICS and MOPTICS follow the same processing order of points. Therefore, for any clustering distance, ε', the resulting clusters are identical and thus the corresponding Omega-Index is 1.

Figure 3(d) shows the time taken to compute the order and reachability distances from the MST computed by MOPTICS compared to the classical OPTICS algorithm for synthetic-random datasets. As mentioned before, this step takes only a fraction of time (on average 0.47%, 0.87%, and 1.65% on synthetic-random, synthetic-cluster, and real-world datasets, respectively) as it only traverses the edges in the MST once.

5.2 POPTICS on Shared Memory

Figure 4 shows the speedup obtained by parallel OPTICS on a shared memory computer (Algorithm 6, denoted by POPTICSS), for various number of threads. The left column in the figure shows the speedup results considering only the local computation stage whereas the right column shows results using total time (local computation and merging) for the three categories of datasets. Clearly, the local computation stage scales well across all the datasets as there is no interaction between the threads. Since local computation takes substantially more time than the merging, the speedup



(a) Local computation (real-world) (b) Total Time (real-world)

(c) Local computation (syn.-rand) (d) Total time (syn.-rand)

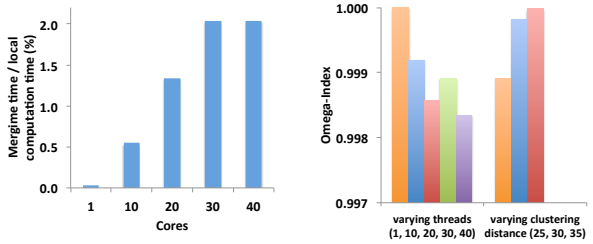
(e) Local computation (syn.-clus) (f) Total time (syn.-clus)

Figure 4: Speedup of parallel OPTICS (Algorithm 6, denoted by POPTICSS) on a shared memory computer. Left column: Local computation in POPTICSS. Right column: Total time (local computation + merging) in POPTICSS.

behavior of just the local computation is nearly identical to that of the overall execution. Note that the speedups for some real-world datasets in Figure 4(a) and 4(b) saturate at around 20 process cores as they are relatively small compared to the other datasets. The maximum speedup obtained in our experiments by POPTICSS is 17.06, 25.62, and 27.50 on *edge*, *r1m*, and *c1.9m*, respectively. However, the ranges of maximum speedup are 5.18 to 17.06 (average 9.93) for real-world, 12.48 to 25.62 (average 19.35) for synthetic-random, and 11.13 to 27.50 (average 20.27) for synthetic-cluster datasets.

Figure 5(a) shows a comparison of the time taken by the merging stage over the local computation stage in percent for the POPTICSS algorithm using dataset $r1.9m$ for various number of process cores. We observe that the merging time remains almost constant (a small fraction of the local computation time on one process core) as the stage is mostly sequential in POPTICSS, and therefore the ratio increases with the number of process cores because the local computation time reduces drastically. Using up to 40 cores, this ratio is maximum 9.60% (average 6.46%), 11.38% (average 5.51%), and 8.01% (average 3.35%) on real-world, synthetic-random, and synthetic-cluster datasets, respectively.

As discussed in Section 4.1, the MST computed by parallel OPTICS could be different than the MST computed by sequential MST-based OPTICS, MOPTICS. Therefore, to compare the quality of the solutions obtained by POPTICSS, we compare the clustering obtained by POPTICSS and classical OPTICS using the Omega-Index. We vary both the number of process cores and clustering distances to observe the tolerance of the POPTICSS algorithm. Figure 5(b) shows the Omega-Index computed on the $c50k$ dataset. The leftmost five bars show the Omega-Index on 1, 10, 20, 30, and 40 process cores, respectively, keeping the clustering distance fixed ($\varepsilon' = 25$). As expected for one core, the Omega-Index is 1, that is, the clusters found by POPTICSS and classical OPTICS match perfectly. Increasing the number of process cores leads to marginal reduction in the Omega-Index, suggesting that the resulting clusters are almost identical compared to ones obtained by the classical OPTICS algorithm. Similarly, varying the clustering distance, ε', (the rightmost three bars in Figure 5(b) representing $\varepsilon' = 25, 30,$ and 35, respectively) keeping the number of cores fixed (30), we also found the Omega-Index close to 1. However, the computation cost for calculating the Omega-Index is high [16] and the available source code [3] we use for this is only capable of dealing with small datasets. We therefore report these numbers for the smallest dataset from each of the three categories.


Mergime time / local computation time (%)
Cores
Omega-Index
varying threads (1, 10, 20, 30, 40)
varying clustering distance (25, 30, 35)

(a) Timing distribution (b) Qualities of the clusters ($c50k$)

Figure 5: (a) Timing distribution for varying number of cores on *r1.9m* and (b) Comparing qualities of the clustering using Omega-Index for varying number of cores and varying clustering distance, e_i on *c50k*

5.3 POPTICS on Distributed Memory

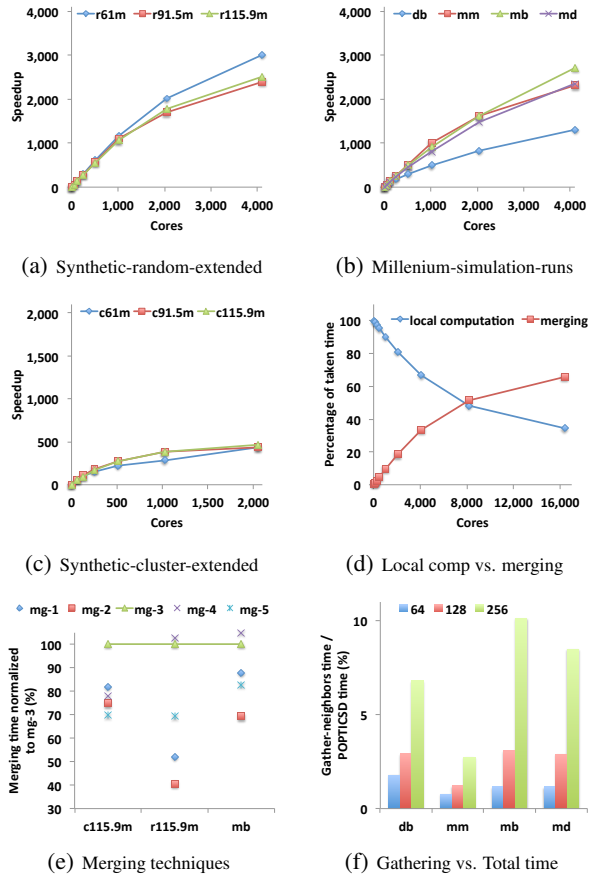
To perform the experiments for parallel OPTICS on a distributed memory computer (POPTICSD), we use Hopper, a Cray XE6 distributed memory parallel computer where each node has two twelve-core AMD MagnyCours 2.1-GHz processors and shares 32 GB of memory. Each core has its own 64 KB L1 and 512 KB L2 caches. Each of the six cores on the MagnyCours processor share one 6 MB of L3 cache. The algorithms have been implemented in C/C++ using the MPI message-passing library and has been compiled with gcc (4.7.2) and -O2 optimization level.

The datasets used in the previous experiments are relatively small for massively parallel computing. We therefore consider a different testbed of 10 datasets, which are again divided into three categories, each with three, three, and four datasets, respectively. The first two categories, called *synthetic-cluster-extended* (*c61m*, *c91.5m*, and *c115.9m*) and *synthetic-random-extended* (*r61m*, *r91.5m*, and *r115.9m*), have been generated synthetically using the IBM synthetic data generator [4, 45]. As the generator is limited to generate at most 2 million high dimensional points, we replicate the same data towards the left and right three times (separating each dataset with a reasonable distance) in each dimension to get datasets with hundreds of million of points. The third category, called *millennium-run-simulation* consists of four datasets from the database on Millennium Run, the largest simulation of the formation of structure with the ΛCDM cosmogony with a factor of 10^{10} particles [32, 49]. The four datasets, MPAGalaxiesBertone2007 (mb) [10], MPAGalaxiesDeLucia2006a (md) [19], DGalaxiesBower2006a (db) [12], and MPAHaloTreesMhalo (mm) [10] are taken from the Galaxy and Halo databases. To be consistent with the size of the other two cat-

egories we have randomly selected 10% of the points from these datasets. However, since the dimension of each dataset is high, we are eventually considering almost a billion floating point numbers. Table 2 shows the structural properties of the datasets and related input parameters. To perform the experiments, we use a parallel *kd-tree* representation as presented in [33, 44] to geometrically partition the data among the processors. However, we do not include the partitioning time while computing the speedup by the POPTICSD.

Table 2: Structural properties of the testbed (synthetic- cluster-extended, synthetic-random-extended, and millennium-run-simulation) and the input parameters, ε and $minpts$, along with the approximate time (in hours) taken by POPTICSD using one process core.

Name	Points	d	ε	$minpts$	Time (**hours**)
$c61m$	61,000,000	10	35	800	9.35
$c91.5m$	91,500,000	10	35	800	10.22
$c115.9m$	115,900,000	10	35	800	13.65
$r61m$	61,000,000	10	45	2	5.72
$r91.5m$	91,500,000	10	45	2	16.57
$r115.9m$	115,900,000	10	45	2	24.55
DGalaxiesBower2006a (db)	101,459,853	8	40	180	20.35
MPAHaloTreesMhalo (mm)	76,066,700	9	40	250	14.66
MPAGalaxiesBertone2007 (mb)	105,592,018	8	40	160	11.74
MPAGalaxiesDeLucia2006a (md)	105,592,018	8	40	150	12.42



(a) Synthetic-random-extended (b) Millenium-simulation-runs

(c) Synthetic-cluster-extended (d) Local comp vs. merging

(e) Merging techniques (f) Gathering vs. Total time

Figure 6: (a)-(c): Speedup of parallel OPTICS (Algorithm 7, denoted by POPTICSD) on a distributed memory computer. (d)-(e): Analyzing POPTICSD.

Figure 6(a), 6(b), and 6(c) show the speedup obtained by algorithm POPTICSD using synthetic-random-extended, millenium-simulation-run, and synthetic-cluster-extended datasets, respectively. Since 64 is the minimum number of process cores we used to perform the experiments for POPTICSD and there is no merging stage in sequential OPTICS, we multiplied the local computation time (taken by POPTICSD on 64 process cores) by 64 and used that value as the approximate sequential time (shown in Table 2) to compute the speedup. We show the speedup using a maximum of 4,096 process cores for the datasets as the speedup saturates and therefore starts decreasing on larger number of process cores. As can be seen the speedup on the synthetic-cluster-extended dataset (Figure 6(c)) is significantly lower than the other two datasets. We observed that the number of edges in the local MSTs on the synthetic-cluster-extended datasets are significantly higher than the synthetic-random-extended and the millenium-simulation-run datasets. On synthetic-cluster-extended, we get a maximum speedup of 466 (average 443) whereas on the synthetic-random-extended and millenium-simula tion-run datasets, these values change to 3,008 (average 2,638) and 2,713 (average 2,169), respectively.

Figure 6(d) shows the trade-off between the local computation and the merging stage by comparing them with the total time (local computation time + merging time) in percent. We use *mb*, one of the millennium-simulation-run dataset for this purpose and continue up to 16,384 process cores to understand the behavior clearly. As can be seen, the communication time increases while the computation time decreases with the number of processors. When using more than 8,000 process cores, communication time starts to dominate the computation time and therefore, the speedup starts being saturated. For example, we achieved a speedup of 2,713 using 4,096 process cores on *mb* whereas the speedup is only 3,952 and 4,750 using 8,192 and 16,384 process cores, respectively. This happens because the overlapping regions among the process cores increase with the increment of the number of process cores. We observe similar behavior for other datasets.

Figure 6(e) compares the variations of the merging stage we used in POPTICSD (Algorithm 7). We call the one presented in Algorithm 7 as *pairwise-merging*, denoted by *mg-3*. Each round in merging traverses the entire graph (i.e. local MSTs) once, thus overhead in merging is proportional to the number of rounds. Therefore, instead of running the merging stage $\log_2 p$ rounds, we terminate the pairwise-merging after 3 rounds and then gather the (so far) merged local MSTs to processor 0 to compute the global MST. We call this variation *mg-4*. Another variation, *mg-5* is the same as *mg-4* except we terminate the pairwise merging when a round takes more than 5 seconds. Two other approaches are based on Boruvka's concept [15], where each processor excludes the edges from their local MSTs that will definitely be part of the global MST. We then use the above two approaches to merge the rest of the edges in the local MSTs. We denote them by *mg-1*, and *mg-2*, respectively. To compare the performance of these variations, we normalized the taken time by the pairwise-merging (*mg-3*) time in percent. Figure 6(e) shows the results using the largest dataset from each category on 64 process cores. We found that in almost all cases, *mg-2* performs the best by taking the minimum time and the reduction is on average 38.46%, 39.52%, and 37.63% on 64, 128, and 256 process cores, respectively. Similar behavior has been found for other datasets. Also note that *mg-1* and *mg-2* are scalable w.r.t. required memory.

Figure 6(f) shows a comparison of the time taken by the gather-neighbors preprocessing step over the total time taken by POPTICSD in percent using 64, 128, and 256 process cores on the millenium-simulation-run datasets. As can be seen, the gather-neighbors step adds an overhead of maximum 4.84% (minimum 0.45%, average 2.14%) of the total time. Similar results have been observed on synthetic-random-extended datasets (maximum 6.38%, minimum

0.77%, average 3.06%). However, these numbers are relatively higher (maximum 26.92%, minimum 6.22%, average 14.46%) on synthetic-cluster-extended datasets as each dataset is much denser, thus the overlapping regions between the processors contain a significant number of points which each processor needs to gather. It is also to be noted that these values increase with the number of processors and also with the ε parameter as the overlapping region among the processors is proportional to them. However, with this scheme the local-computation stage in POPTICSD can perform the processing without any communication overhead similar to POPTICSS. The alternative would be to perform communication for each point to obtain its remote neighbors.

6. CONCLUSION AND FUTURE WORK

In this study we have revisited the well-known density based clustering algorithm, OPTICS. This algorithm is known to be challenging to parallelize as the computation involves strongly inherent sequential data access order. We present a scalable parallel OPTICS (POPTICS) algorithm designed using graph algorithmic concepts. More specifically, we exploit the similarities between OPTICS and PRIM's Minimum Spanning Tree (MST) algorithm. Additionally, we use the disjoint-set data structure to extract the clusters in parallel from the MST for increasing concurrency. POPTICS is implemented using both OpenMP and MPI. The performance evaluation used a rich set of high dimensional data consisting of instances from real-world and synthetic datasets containing up to a billion floating point numbers. Our experimental results conducted on a shared memory computer show scalable performance, achieving speedups up to a factor of 27.5 when using 40 cores. Similar scalability results have been obtained on a distributed-memory machine with a speedup of 3,008 using 4,096 process cores. Our experiments also show that while achieving the scalability, the quality of the results given by POPTICS is comparable to those given by the classical OPTICS algorithm. We intend to conduct further studies to provide more extensive results on much larger number of cores with datasets from different scientific domains. Finally, we note that our algorithm also seems to be suitable for other parallel architectures such as GPU and heterogenous architectures.

7. ACKNOWLEDGMENTS

This work is supported in part by the following grants: NSF awards CCF-0833131, CNS-0830927, IIS-0905205, CCF-0938000, CCF-1029166, and OCI-1144061; DOE awards DE-FG02-08ER25 848, DE-SC0001283, DE-SC0005309, DESC0005340, and DESC0 007456; AFOSR award FA9550-12-1-0458. This research used Hopper Cray XE6 computer of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

8. REFERENCES

[1] Parallel K-means data clustering, 2005. http://users.eecs.northwestern.edu/ wkliao/Kmeans/.

[2] CLUTO - clustering high-dimensional datasets, 2006. http://glaros.dtc.umn.edu/gkhome/cluto/cluto/.

[3] Cliquemod, 2009. http://www.cs.bris.ac.uk/ steve/networks/cliquemod/.

[4] R. Agrawal and R. Srikant. Quest synthetic data generator. *IBM Almaden Research Center*, 1994.

[5] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD*, pages 49–60, New York, NY, USA, 1999. ACM.

[6] D. Arlia and M. Coppola. Experiments in parallel clustering with DBSCAN. In *Euro-Par 2001 Parallel Processing*, pages 326–331. Springer, LNCS, 2001.

[7] H. Backlund, A. Hedblom, and N. Neijman. A density-based spatial clustering of application with noise. 2011. http://staffwww.itn.liu.se/ aidvi/courses/06/dm/Seminars2011.

[8] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *Proceedings of the 1990 ACM SIGMOD*, 19(2):322–331, 1990.

[9] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] S. Bertone, G. De Lucia, and P. Thomas. The recycling of gas and metals in galaxy formation: predictions of a dynamical feedback model. *Monthly Notices of the Royal Astronomical Society*, 379(3):1143–1154, 2007.

[11] D. Birant and A. Kut. ST-DBSCAN: An algorithm for clustering spatial-temporal data. *Data & Knowledge Engineering*, 60(1):208–221, 2007.

[12] R. Bower, A. Benson, R. Malbon, J. Helly, C. Frenk, C. Baugh, S. Cole, and C. Lacey. Breaking the hierarchy of galaxy formation. *Monthly Notices of the Royal Astronomical Society*, 370(2):645–655, 2006.

[13] S. Brecheisen, H. Kriegel, and M. Pfeifle. Parallel density-based clustering of complex objects. *Adv. in Know. Discovery and Data Mining*, pages 179–188, 2006.

[14] M. Chen, X. Gao, and H. Li. Parallel DBSCAN with priority r-tree. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 508–511. IEEE, 2010.

[15] S. Chung and A. Condon. Parallel implementation of bouvka's minimum spanning tree algorithm. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 302–308. IEEE, 1996.

[16] L. M. Collins and C. W. Dent. Omega: A general formulation of the rand index of cluster recovery suitable for non-disjoint solutions. *Multivariate Behavioral Research*, 23(2):231–242, 1988.

[17] M. Coppola and M. Vanneschi. High-performance data mining with skeleton-based structured parallel programming. *Parallel Computing*, 28(5):793–813, 2002.

[18] T. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[19] G. De Lucia and J. Blaizot. The hierarchical formation of the brightest cluster galaxies. *Monthly Notices of the Royal Astronomical Society*, 375(1):2–14, 2007.

[20] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data mining*, volume 1996, pages 226–231. AAAI Press, 1996.

[21] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996.

[22] M. Forina, M. C. Oliveros, C. Casolino, and M. Casale. Minimum spanning tree: ordering edges to identify clustering structure. *Analytica Chimica Acta*, 515(1):43 – 53, 2004.

[23] Y. Fu, W. Zhao, and H. Ma. Research on parallel DBSCAN algorithm design based on mapreduce. *Advanced Materials Research*, 301:1133–1138, 2011.

[24] B. Galler and M. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7:301–303, 1964.

[25] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 18(1):pp. 54–64, 1969.

[26] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan Kaufmann, 2011.

[27] H. Kargupta and J. Han. *Next generation of data mining*, volume 7. Chapman & Hall/CRC, 2009.

[28] M. B. Kennel. KDTREE 2: Fortran 95 and C++ software to efficiently search for near neighbors in a multi-dimensional Euclidean space, 2004. Institute for Nonlinear Science, University of California.

[29] H.-P. Kriegel and M. Pfeifle. Hierarchical density-based clustering of uncertain data. In *Data Mining, Fifth IEEE International Conference on*, pages 4–pp. IEEE, 2005.

[30] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, Feb. 1956.

[31] L. Lelis and J. Sander. Semi-supervised density-based clustering. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 842–847. IEEE, 2009.

[32] G. Lemson and the Virgo Consortium. Halo and galaxy formation histories from the millennium simulation: Public release of a VO-oriented and SQL-queryable database for studying the evolution of galaxies in the LambdaCDM cosmogony. *Arxiv preprint astro-ph/0608019*, 2006.

[33] Y. Liu, W.-k. Liao, and A. Choudhary. Design and evaluation of a parallel HOP clustering algorithm for cosmological simulation. In *Proceedings of IPDPS 2003*, page 82.1, Washington, DC, USA, 2003. IEEE.

[34] Z. Lukić, D. Reed, S. Habib, and K. Heitmann. The structure of halos: Implications for group and cluster cosmology. *The Astrophysical Journal*, 692(1):217, 2009.

[35] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. USA, 1967.

[36] S. Madeira and A. Oliveira. Biclustering algorithms for biological data analysis: a survey. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 1(1):24–45, 2004.

[37] F. Manne and M. Patwary. A scalable parallel union-find algorithm for distributed memory computers. In *Parallel Processing and Applied Mathematics*, pages 186–195. Springer, LNCS, 2010.

[38] A. Mukhopadhyay and U. Maulik. Unsupervised satellite image segmentation by combining SA based fuzzy clustering with support vector machine. In *Proceedings of 7th ICAPR'09*, pages 381–384. IEEE, 2009.

[39] G. Murray, G. Carenini, and R. Ng. Using the omega index for evaluating abstractive community detection. In *Proceedings of Workshop on Evaluation Metrics and System Comparison for Automatic Summarization*, pages 10–18, Stroudsburg, PA, USA, 2012.

[40] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 205–214. ACM, 2012.

[41] H. Park and C. Jun. A simple and fast algorithm for K-medoids clustering. *Expert Systems with Applications*, 36(2):3336–3341, 2009.

[42] M. Patwary, M. Ali, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 827–835. IEEE, 2012.

[43] M. Patwary, J. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA 2010)*, pages 411–423. Springer, LNCS 6049, 2010.

[44] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 62:1–62:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[45] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 3.0. Technical report, Technical Report CUCIS-2005-08-01, Northwestern University, 2010.

[46] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

[47] R. Setia, A. Nedunchezhian, and S. Balachandran. A new parallel algorithm for minimum spanning tree problem. In *Proc. International Conference on High Performance Computing (HiPC)*, pages 1–5, 2009.

[48] G. Sheikholeslami, S. Chatterjee, and A. Zhang. WaveCluster: a wavelet-based clustering approach for spatial data in very large databases. *The VLDB Journal*, 8(3):289–304, 2000.

[49] V. Springel, S. White, A. Jenkins, C. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, et al. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 435(7042):629–636, 2005.

[50] M. Surdeanu, J. Turmo, and A. Ageno. A hybrid unsupervised approach for document clustering. In *Proceedings of the 11th ACM SIGKDD*, pages 685–690. ACM, 2005.

[51] R. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979.

[52] W. Wang, J. Yang, and R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proceedings of the International Conference on Very Large Data Bases*, pages 186–195. IEEE, 1997.

[53] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: the state of the art and comparative study. *ACM Computing Surveys*, 45(4), 2013.

[54] X. Xu, J. Jäger, and H. Kriegel. A fast parallel clustering algorithm for large spatial databases. *High Performance Data Mining*, pages 263–290, 2002.

[55] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD Record*, volume 25(2), pages 103–114. ACM, 1996.

[56] A. Zhou, S. Zhou, J. Cao, Y. Fan, and Y. Hu. Approaches for scaling DBSCAN algorithm to large spatial databases. *Computer science and technology*, 15(6):509–526, 2000.